Extended recursion-based formalization of virus mutation

Philippe Beaucamps*
Loria, France
philippe.beaucamps_ at_ loria_ dot_ fr

July 6, 2008

Abstract

Computer viruses are programs that can replicate themselves by infecting other programs in a system. Bonfante, Kaczmarek and Marion have recently proposed a classification of viruses which relies on the recursion theory and its recursion theorems. We propose an extension of their formalism to consider in a more practical way the mutation of viruses. In particular, we are interested in modelling any depth of mutation, not just the first two levels. We show that this formalism still relies on recursion theorems, whatever the depth of mutation, even in the case of infinite depth. We also extend furthermore this formalism to model the viability of viral replication, which ensures that an infected program still can propagate the virus. An application of the proposed formalism to the class of combined viruses (multi-part viruses) is studied. Finally, given that metamorphic viruses can be modelled by grammars operating on grammars, we study a recursion-based approach of formal grammars and show that the recursion theorems of the recursion theory can be ported to the formal grammars theory.

Keywords: *Computer viruses, virus mutation, polymorphism, metamorphism, recursion theory, formal grammars, combined viruses*

1 Introduction

Computer infections are a serious concern in nowadays IT infrastructures. These infections are carried out using miscellaneous types of malware, among which computer viruses: such programs replicate themselves in a host environment, possibly mutating during the replication and possibly carrying a payload. These viruses have been modelled very early by F. Cohen [Coh86] using Turing machines, and then by Adleman [Adl88] using recursive functions. Lately, Filiol [Fil05] and Bonfante, Kaczmarek and Marion [BKM05, BKM07] have proposed a new formalization of computer viruses which encompasses any previous

*also École Supérieure et d'Application des Transmissions, Rennes, France.

approach and allows a classification where the existence of each class relies on a variant of Kleene's recursion theorem [Kle38].

The major stake in detecting viruses is virus mutation. Simple viruses are detected by pattern-matching. However, some viruses mutate their code along any replication: polymorphic viruses encrypt their code and mutate the decryption function only, whereas metamorphic viruses mutate the whole code. Thus simple polymorphic viruses always replicate using the same code: their mutation function is fixed. Metamorphic viruses however mutate their code and thus are able to mutation their mutation function.

In this paper, starting from the work of Bonfante & al, we adopt a more practical approach by considering directly in our formalism these mutation functions. However, rather than limiting ourselves to one mutation function, we hypothetically consider the case of mutation at any depth. We study this formalism according to two approaches, one being behavioural, which corresponds to Bonfante & al's work, the other one being syntactic. After formalizing these approaches, we consider the case of infinite depth of mutation and conclude with the problem of viability of the replication: how do we ensure that an infected program continues replication. This formalism is finally illustrated by the case of combined viruses, which are multi-part viruses.

Poly/metamorphic viruses can also be modelled using formal grammars [Fil07]. Metamorphic viruses are in particular modelled by grammars operating on other grammars: the parallel with recursive functions seen as integers operating on integers is straightforward. Thus we investigate in the end a recursion approach of the theory of formal grammars.

2 Viruses in the recursion theory

2.1 Notations

In the recursion theory, programs are represented by integers (using Gödel's numbering). For a given program $p \in \mathbb{N}$, φ_p is the semi-recursive function computed by p. Encoding of tuples of integers into integers is denoted by $\langle \cdots \rangle$. When unambiguous, brackets may be omitted.

2.2 Recursion theorems

Self-reproduction of programs relies on two fundamental theorems, established by Kleene [Kle38]:

Theorem 1 (Iteration theorem)**.** *There exists a semi-recursive function* $\mathcal{S} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *which verifies:*

For any program p*, for any integer* x*, the program* $\mathcal{S}(p, x)$ *verifies:*

$$\forall y \in \mathbb{N},\ \varphi_{\mathcal{S}(p,x)}(y) = \varphi_p(x, y)$$

$\mathcal{S}(p, x)$ *is said to specialize program* p *on input* x*.*

$\mathcal{S}$ is called the iteration function or the s-m-n function.

Theorem 2 (Recursion theorem)**.** *For any recursive function f, there exists a program e such that:*

$$\forall x \in \mathbb{N},\ \varphi_e(x) = f(e, x)$$

This theorem proves the existence of self-reproducing programs. For instance, Quine programs[1] merely correspond to the function: $f(p, x) = p$.

Subsequently, Smullyan extended the recursion theorem to two recursive functions [Smu93]:

Theorem 3 (Double recursion theorem)**.** *For any recursive functions f and g, there are programs e_1 and e_2 such that:*

$$\begin{aligned} \forall x, \quad \varphi_{e_1}(x) &= f(e_1, e_2, x) \\ \varphi_{e_2}(x) &= g(e_1, e_2, x) \end{aligned}$$

These theorems, along with their variants, provide a basis to Bonfante & al's formalism, as detailed in the next section.

2.3 Current formalism

Bonfante, Kaczmarek and Marion defined a virus with respect to a semi-recursive function $\mathcal{B}$, which is called propagation function [BKM07]. This function describes how a virus can infect (insert itself into) a program. We here recall the different classes of viruses they defined and their associated results.

Definition 1 (Virus)**.** *A program v is a virus wrt a semi-recursive function $\mathcal{B}$ iff:*

$$\forall p, \forall x, \quad \varphi_v(p, x) = \varphi_{\mathcal{B}(v,p)}(x)$$

Existence of such viruses comes from a simple application of Kleene's recursion theorem. Since the proof of this theorem is constructive, a virus can be constructed for any propagation function [BKM07].

Moreover, Bonfante & al proved that this generic definition encompasses any previous definition of viruses by Cohen [Coh86], Adleman [Adl88] and Zuo and Zhou [ZZ04].

2.3.1 Blueprint viruses

A blueprint virus [BKM07] is defined wrt a semi-recursive function g which specifies the behaviour of the virus in an environment. Such viruses simply duplicate their code when replicating.

Definition 2 (Blueprint virus)**.** *A program p is a blueprint virus wrt a semi-recursive function g iff:*

[1] A Quine program is a program that outputs its own code.

- *v is a virus wrt some propagation function.*
- $\forall p, x, \quad \varphi_v(p, x) = g(v, p, x)$

Definition 3 (Distribution engine)**.** *A semi-recursive function d_v is a distribution engine iff there exists a fixed propagation function $\mathcal{B}$ such that, for any i, $d_v(i)$ is a virus wrt $\mathcal{B}$.*

Bonfante & al. show that there exists a *blueprint distribution engine* which yields a blueprint virus for any semi-recursive function g and wrt a fixed propagation function, which happens to be the iteration function $\mathcal{S}$.

In order to allow the mutation of blueprint viruses during replication, evolving blueprint viruses are defined:

Definition 4 (Evolving blueprint virus)**.** *A program d_v is a distribution of evolving blueprint viruses wrt a semi-recursive function g iff:*

- *d_v is a distribution engine.*
- $\forall i, p, x, \quad \varphi_{d_v(i)}(p, x) = g(d_v, i, p, x)$

The existence of such viruses relies on a parameterized variant of Kleene's recursion theorem.

2.3.2 Smith viruses

Evolving blueprint viruses are defined wrt a fixed propagation function. We now define smith viruses wrt a specification function which depends on their propagation function. Thus a smith virus corresponds to the couple of the virus and its propagation function:

Definition 5 (Smith virus)**.** *Two programs v and $\mathcal{B}$ are a smith virus iff:*

- *v is a virus wrt $\mathcal{B}$*
- $\forall p, x, \quad \varphi_v(p, x) = g(\mathcal{B}, v, p, x)$

Existence of smith viruses relies on the double recursion theorem (theorem 3).

Definition 6 (Virus distribution)**.** *A virus distribution is a pair $(d_v, d_\mathcal{B})$ such that for any i,$\varphi_{d_v}(i)$ is a virus wrt $\varphi_{d_\mathcal{B}}(i)$.*

Again, as for blueprint distribution engines, there exist smith virus distributions which are virus distributions operating on specification functions and yielding smith viruses wrt these specification functions.

Finally, the class of smith distributions is defined by the viruses which can mutate their code along with their propagation function (metamorphic viruses):

Definition 7 (Smith distribution)**.** *Two programs d_v and $d_\mathcal{B}$ are a smith distribution wrt a semi-recursive function g iff:*

- $(d_v, d_{\mathcal{B}})$ *is virus distribution.*
- $\forall i, p, x, \quad \varphi_{\varphi_{d_v}(i)}(p, x) = g(d_{\mathcal{B}}, d_v, i, p, x)$

Existence of such viruses relies on a parameterized version of the double-recursion theorem.

3 Recursion and vertical mutation

3.1 Vertical mutation chains

First, let's consider the seeming equivalence between blueprint viruses and smith distributions. A blueprint virus (along with its propagation function) can be seen as a smith distribution, with constant virus distribution. Same goes for evolving blueprint viruses. Conversely, a smith distribution can be seen as a distribution of evolving blueprint viruses. Let $(d_v, d_{\mathcal{B}})$ be a smith distribution wrt a specification function g: each virus generated by d_v is a virus wrt its own propagation function. However, if we consider the semi-recursive function g' defined by the specialization of g for $d_{\mathcal{B}}$ ($g' = \mathcal{S}(g, d_{\mathcal{B}})$), then d_v is an evolving blueprint virus distribution wrt g' and the propagation function $\mathcal{S}$ (iteration function). Thus the classes of evolving blueprint viruses and of viruses generated by smith distributions are formally identical.

Moreover, the proposed formalism only considers two levels of mutation: a given virus can mutate its code and its propagation function. We thus extend this formalism to model any depth of mutation. This mutation is vertical, as opposed with horizontal mutation which occurs on a given depth of mutation between different virus generations.

Let's call mutation function at level n the function that models the mutation of the $n-1$-mutation function, given an environment and mutation functions at lower levels. At level 0, the mutation function yields the infected program when given as input the virus, a target program and an environment. These functions will be formally defined later.

We are also interested in the number of mutation levels from which the mutation functions can be considered fixed and we will more particularly study the case of infinite (vertical) mutation chains, as well as the notion of viable replication (i.e. an infected program can still effectively replicate).

From a syntactic perspective, let's now suppose that a virus has no access to its propagation function: then considering this propagation function isn't justified in a sense and we could consider that this propagation function is the iteration function. So the number of mutation levels is motivated by the actual ability to extract the mutation function on any of these levels. Similarly viruses that mutate their code in a fixed way can actually be considered as strictly mutating their mutation function. For instance, consider a virus v_0 which yields its own code (using a self reference provided by the environment) plus a space, and a virus v_1 which is a variant of a Quine program (a program that outputs

its own code) modified in such a way that it appends a space at the end of its code. v_0 and v_1 have then the same behaviour when replicating, but v_0 has a fixed mutation function whereas v_1 has a variable mutation function since it actually depends on the current virus code.

3.2 Notations

Let v be a given virus, p a program to infect and x an environment. In the following, when program p and environment x are unambiguous, we will denote by v' the result of infection of program p by virus v in environment x (i.e. the resulting infected program).

We recursively define the mutation functions of the virus v by:

$$\begin{aligned} \mu_{0,v}(v,p,x) &= v' \\ \forall i, \quad \mu_{i,v}(\mu_{i-1,v},\cdots,\mu_{0,v},p,x) &= \mu_{i-1,v'} \end{aligned}$$

For sake of clarity, we may denote by ϕ_v the ground level mutation function and by ψ_v the level 1 mutation function.

3.3 Behavioural and syntactic equations

The following results respond to two of the previous questions. When can we consider that a mutation function is fixed? And, supposing we can consider that a mutation function is fixed, on what basis should we actually consider that it is not? The first question will explain the prior considerations on evolving blueprint viruses and smith distributions, while the second question will make more explicit the reasons why in some cases it remains interesting to consider the behaviour of mutation functions. Having answered these questions, we can formalize in more details the mutation on any level.

Mutation functions can be studied from two approaches: a behavioural one and a syntactic one. The behavioural approach corresponds to the one adopted by Bonfante & al.

Lemma 1. *Let n be a given depth. If there exists a recursive function g_{n-1} such that:*

$$\forall v, \quad g_{n-1}(v) = \mu_{n-1,v}(\mu_{n-2,v},\cdots,\mu_{0,v},v)$$

Then:

1. *There exists a* fixed *mutation function μ_n such that, for any virus v (usually of a given strain), its mutation function $\mu_{n-1,v}$ mutates according to μ_n.*
2. *Any deeper mutation function is fixed, being equal to the identity.*

Proof. $\mu_{n,v}$ verifies:

$$\forall v,p,x \quad \mu_{n,v}(\mu_{n-1,v},\cdots,\mu_{0,v},v,p,x) = \mu_{n-1,\mu_{0,v}(v,p,x)} = \mu_{n-1,v'}$$

The new mutation function must be valid wrt the infected form of the virus, v', which is expressed by:

$$\begin{aligned}
&\forall v, p, x \\
&\qquad \mu_n(\mu_{n-1,v}, \cdots, \mu_{0,v}, v, p, x)(\mu_{n-2,v'}, \cdots, \mu_{0,v'}, v') \\
&= \quad \mu_{n-1,v'}(\mu_{n-2,v'}, \cdots, \mu_{0,v'}, v') \\
&= \quad g_{n-1}(\mu_{0,v}(v, p, x))
\end{aligned}$$

Since the constraints on μ_n are local (for a given v, μ_n must yield a function that is valid at least on the v' specific input), taking $\mu_n(\mu_{n-1,v}, \cdots, \mu_{0,v}, v, p, x) = \mu_{n-2,w}, \cdots, \mu_{0,w}, w \mapsto g_{n-1}(\mu_{0,v}(v, p, x))$ ends the proof. □

This lemma allows us to consider relations that *characterize the* local *behaviour of a mutation function*, that is equations expressing that a given function locally behaves as the considered mutation function. If such a characterization exists, then it is represented by the function g_{n-1}. For instance, in the case of the ground level mutation function ϕ_v, we can characterize this function by the relation: $\phi_v(v) = \pi_1 \circ v$, which corresponds to the function $g_0 = v \mapsto \pi_1 \circ v$ (where π_1 corresponds to the projection on the first component, assuming that this component contains the infected program). Then lemma 1 tells us that the first level mutation function ψ_v can be considered fixed. This result has a local extent, that is wrt the propagation. If for instance we are also able to characterize the result of the mutation function ψ_v with respect to a virus v, then the previous result would be discarded. Yet it remains locally valid, which amounts to the following consistency property:

$$\forall v, p, x, \quad g_1(v)(p, x)(v') = g_0(v')$$

Thus, on a strictly functional perspective, we can consider a single level of mutation, as deeper mutation functions can be approximated. Nevertheless, in general, it makes sense to consider the mutation of ϕ, since g_0 is defined by: $g_0 = v \mapsto \pi_1 \circ v$.

As was previously explained, we also want to consider the case of the mutation functions being explicitly and syntactically enclosed into (and thus extractable from) the virus. Then we would like to relate both perspectives and make them compatible with each other. This second case leads to the following lemma (derived from lemma 1):

Lemma 2. *Let n be a given depth. If there exist two recursive functions h_0 and h_{n-1} such that:*

$$\forall v, \quad h_0(v) = \mu_{0,v} \text{ and } h_{n-1}(v) = \mu_{n-1,v}$$

Then:

1. *There exists a* fixed *mutation function μ_n such that, for any virus v (usually of a given strain), its mutation function $\mu_{n-1,v}$ mutates according to μ_n.*

2. Any deeper mutation function is fixed, being equal to the identity.

Proof. Simply define μ_n as:

$$\forall v, p, x \quad \mu_n(\mu_{n-1,v}, \cdots, \mu_{0,v}, v, p, x) = h_{n-1}(h_0(v)(v, p, x))$$

□

Thus functions h_i are similar to functions g_i but operate at a deeper level and no longer on a local scale. Rather than characterizing the behaviour of mutation functions wrt the behaviour of the virus, they characterize the fact that mutation functions can be syntactically and *globally* extracted from the virus. This is the case for instance of viruses where the mutation grammars of level 1 and possibly deeper are directly encoded into the data of the virus, allowing us to define h_0, h_1, etc. Thus, for a given virus strain, there is no limit to the depth of mutation we should consider, since any mutation function at any level could be hard-coded into the virus.

Both perspectives yield consistent equations.

Given the recursive functions g_i, we get the following behavioural equations:

$$\forall p, x \quad \mu_{i,v}(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x) = g_i(v)(p, x)$$

Also, given the recursive functions h_i, we get the following syntactic equations:

$$\forall p, x \quad \mu_{i,v}(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x) = h_i(v)(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x)$$

We finally redefine[2] our original equation on v, for a given depth $n-1$:

$$\forall p, x \quad v(p, x) = f(\mu_{0,v}, \cdots, \mu_{n-1,v}, v, p, x)$$

Then application of the $(n+1)$-ary recursion theorem (see appendix B.1) to these equations, in any perspective, entails the existence of v and of such mutation functions.

Thus the first perspective entails the existence of the mutation functions but at a limited level as it is related to the characterization of the corresponding mutation functions. Deeper mutation functions must be approximated by fixed ones. And the second perspective also entails existence of the mutation functions, this time at any level – as long as the corresponding mutation function can be extracted from the virus – but then there is no proof that the mutation functions are locally compatible with the actual ones. Thus, to make both perspectives compatible with each other, we simply add the following local

[2]Note that this equation is furthermore justified by the fact that existence of these functions g_i or h_i relies precisely on the ability of the virus to be able to access and alter its mutations functions, thereby justifying the dependency of f on those.

constraints on the h_i functions:

$$\begin{aligned} h_0(v)(v) &= g_0(v) \\ &= \pi_1 \circ v \\ \forall p, x, \quad h_i(v)(\mu_{i-1,v}, \cdots, \mu_{0,v}, v, p, x) &= \mu_{i-1,v'} \\ &= h_{i-1}(h_0(v)(v, p, x)) \end{aligned}$$

These constraints are common sense as the h_i functions could return anything unrelated to the mutation functions. Supposing the ground level mutation grammar is encoded into the virus, then this constraint simply requires that the grammar returned by h_0 is the grammar being actually used to mutate the virus.

The original propagation function concept was thus extended by a more general consideration of mutation functions at any level, whereas the requirement of a correlation between a virus and its propagation function, as expressed in the original definition[3], is now an intuitive formulation of the characterization of a mutation function with respect to a virus. The latter approach also allows to directly infer these mutation functions from the virus. Although that inference is easily understood in the case of the ground level mutation function ϕ, as it can be computed directly from the execution of a virus in a controlled environment, it mostly depends on the virus internal (programming) structure for deeper levels.

These results, that require an analysis of viruses from a more syntactic (implementation related) perspective, motivate their study from a grammar perspective, though some concepts are still easier to comprehend from the recursion theory perspective.

3.4 Infinite Vertical Mutation Chains

Finally, we might want to consider the case of an infinite vertical mutation chain – i.e. in the mutation functions. As was shown previously, no limit can be enforced on the depth of mutation. However, apart from the practice where mutations are usually limited to the first two levels, the case of an infinite set of mutations in the mutation functions is interesting to consider, with regards to its consistence as well as its theoretical basis. One can actually show that, using the previous equations and a countable version of the recursion theorem (see appendix B.2), we are able generalize the previous results to any number of mutation functions. Indeed, this theorem entails the existence of a countable sequence of mutation functions that follow the previous specifications.

Thus, although the previous results were corroborated by the existence of actual implementations and thereby provided a theoretical background to these ones, this precise result actually shows that, even though there is currently no implementation of a virus with an infinite vertical mutation chain, such viruses theoretically do exist. Their practical existence is an open problem.

[3] namely: $\forall p, x,\ v(p, x) = \mathcal{B}(v, p)(x)$

Also, when considering these mutation functions on a vertical scale, one could wonder if this does not actually correspond to a recursion structure, on a higher abstraction level. Indeed, for any finite number of mutation functions, the multary recursion theorem is derived from the basic recursion theorem and remains on an horizontal scale. Looking at the countable recursion theorem and its proof, one can actually see that it precisely corresponds to moving to a 1-higher abstraction level: the proof considers semi-recursive functions F and E that operate directly on the space of mutation functions. Then the recursion theorem is applied in this dimension. Thus the basic recursion theorem manipulates functions, while the countable recursion theorem manipulates sets of functions, and one could even go further in the abstraction levels.

And necessarily, the previous remarks raise the question of a new recursion level that would operate directly on the scale of those F and E functions. This has not been investigated in this article.

3.5 Viable replication

To conclude with this formalism, we consider the problem of viable replication: how to make sure that the mutated form of a virus will continue replication. This is the very basis of virus theory. The case of basic viruses that simply replicate by copying themselves is straightforward. However mutating viruses do not anymore verify the equations that gave birth to their strains. Though this is not explicitly mentioned in Bonfante & al.'s article [BKM07], they bring an answer for the case $n < 2$ with the evolving blueprint viruses and the smith distributions. We merely generalize their result to the previous formalism, for any depth of mutation, including infinite depth.

Since the replication is linear, and rather than adding extra-requirements, Bonfante & al. index the viruses by a parameter i: thus all mutated forms of a virus are gathered into a so-called distribution engine, as explained in section 2.3. Then the recursion theorem is applied on this distribution engine rather than on a given virus. In a sense, this is an application of the countable recursion theorem to the countable set of all mutated forms of the virus. Such a distribution engine can be generalized to take into account any depth of mutation. Let's denote by d_v the distribution engine of v and by d_μ^j the distribution engine of the mutation function $\mu_{j,*}$: $d_\mu^j(i) \equiv \mu_{j,d_v(i)}$.

Then the equations these distributions must verify are the following:

$$\begin{aligned}
\forall p, x \quad d_v(i,p,x) &= f(d_\mu^0, \cdots, d_\mu^{n-1}, d_v, i, p, x) \\
\forall p, x \quad d_\mu^0(d_v,i,p,x) &= f_0(d_\mu^0, \cdots, d_\mu^{n-1}, d_v, i, p, x) \\
&\cdots \\
\forall p, x \quad d_\mu^{n-1}(d_\mu^{n-2}, \cdots, d_\mu^0, d_v, i, p, x) &= f_{n-1}(d_\mu^0, \cdots, d_\mu^{n-1}, d_v, i, p, x)
\end{aligned}$$

where the functions f_i are functions g_i or h_i from lemmas 1 and 2 (behavioural and syntactic functions).

Thus the $(n+1)$-ary recursion theorem still applies. The same goes for an infinite depth of mutation.

3.6 Application: combined viruses

Combined viruses, also called k-ary viruses [Fil07], are a particular class of viruses that are composed of several parts, which operate together, in a sequential or parallel way. Filiol decomposed these viruses into several classes [Fil07], depending on whether they operate independently (without any references to each other) or not. Class A contains strongly dependent codes, class B contains independent codes and class C contains weakly dependent codes (one-way dependency). Such viruses, whatever their class, are not compatible at first sight with our previous model.

Each virus part v_i might behave according to its own mutation function f_i. Thus each part might have its own independent horizontal and vertical mutation chain. Fully independent combined viruses are the simplest case: they correspond to the action of independent viruses. We will consider the two following cases:

3.6.1 Class B viruses – independent parts

First we shall note that a combined virus can be made of k parts and replicate into k' parts, which prevents us from considering mutation functions on the scale of each part. In the present case, the f_i functions have two arguments: the part v_i and the environment p, x that we will denote by x for sake of simplicity. However they must be considered as taking part to interactions with the other parts: depending on the virus, a part may be waiting for another part to complete a task or to answer a query. Consequently, we will consider the functions f_i^* that take a third and fourth argument, namely the execution state (subsequently denoted by j), which allows to resume function f_i at any stage of its execution, and a number of execution steps (subsequently denoted by n) to perform before being suspended. We could consider this execution state to be the instruction pointer `eip` (along with viral data contained in other registers and the memory). Repeated application of Kleene's recursion theorem now yields:

$$\exists v_1^*, \forall j, n, x, \quad v_1^*(j, n, x) = f_1^*(v_1^*, j, n, x) \tag{1a}$$

$$\cdots$$

$$\exists v_k^*, \forall j, n, x, \quad v_k^*(j, n, x) = f_k^*(v_k^*, j, n, x) \tag{1k}$$

Then the viral part v_i is simply defined by: $\forall x, v_i(x) = v_i^*(0, \infty, x)$, where 0 represents the initial execution state.

Execution of the combined virus $v = \{v_1, \cdots, v_k\}$ on an environment x can be represented by an execution sequence: $E(v, x) = \mathbb{N} \mapsto$ Steps, where Steps

is defined by: Steps $= \{\langle i, j, n, x' \rangle \, | i \in [1, \cdots, k], j, n \in \mathbb{N}, x' \in \text{Env}\}$. i is the index of the part to be executed, j is the execution state it will start at, n is the number of execution steps to perform, and x' is the environment it will be executed into. We do not detail the consistence properties like j_s being required to match the last j_e of the current part (or 0 on the first execution) and similar sequence properties on x'.

Let's denote by $v(x)$ the result of the execution of v on environment x and suppose that (where of course $x_0 = x$):

$$E(v, x) = (\langle i_0, j_0, n_0, x_0 \rangle, \langle i_1, j_1, n_1, x_1 \rangle, \cdots, \langle i_m, j_m, n_m, x_m \rangle)$$

Then:

$$v(x) = v^*_{i_m}(j_m, n_m, v^*_{i_{m-1}}(j_{m-1}, n_{m-1}, \cdots v^*_{i_0}(j_0, n_0, x) \cdots))$$

The miscellaneous interruptions are either the result of manual ones or the result of interactions with the environment like waiting for resources or for a response to a query, etc.

Finally, we represent this global interaction process as the result of an interaction function f which, given the k viral parts, represents the result of the execution of v on an environment x. Since no physical entity is associated to the global virus v, this function f can only consist of executing a part, interrupting it, executing another one, interrupting it, resuming the first one, and so on. Thus this function f is merely the execution function associated to the execution sequence of the virus. We suppose that this execution sequence is normalized in the sense that a viral part is executed until it is automatically interrupted because of a resource need. We express the viral property of v by:

$$v(x) = f(v_1, \cdots, v_k, x) \tag{2}$$

Since f consists of the action of a given part, followed by the action of another part and so on, we have:

$$f(v_1, \cdots, v_k, x) = f^*_{i_m}(v^*_{i_m}, j_m, n_m, f^*_{i_{m-1}}(v^*_{i_{m-1}}, j_{m-1}, n_{m-1}, \cdots f^*_{i_0}(v^*_{i_0}, j_0, n_0, x))$$

Using the previous equations 1a-1k, one can then easily verify that equation 2 is verified. Note that this result directly comes from the very restrictive design of f, which models the behaviour of v.

Other abstractions have also been studied that try to reconcile the theories of recursive functions and of interaction [JFD07]. Although they would be interesting to investigate with respect to our model, our current choice is only motivated by the simplicity of the present abstraction with regards to our problem.

Consideration of the mutation functions is a bit more tricky. First, we have to review our definition of $\mu_{0,v} \equiv \phi_v$. As told previously, in the general case, the mutation function only makes sense on the scale of the whole virus. So if v

replicates into v', we want: $\mu_{0,v}(v,x) = v'$, where v and v' are actually multi-part viruses. As for the case of simple viruses, v' can be computed from the execution of v. The number of parts depends on v and the environment only (whether this number is randomly generated or not). Let κ denote the function returning the new k' from the current virus and the environment: $\kappa(v,x) = |v'|$. Then, with the same simplification as in the previous sections (for common viruses):

$$\mu_{0,v}(v,x) = \langle \pi_1(v(x)), \cdots, \pi_{\kappa(v,x)}(v(x)) \rangle$$

In other words: $g_0 = v \mapsto x \mapsto \langle \pi_1(v(x)), \cdots, \pi_{\kappa(v,x)}(v(x)) \rangle$, where g_0 is the function defined in lemma 1.

Deeper mutation functions are unchanged, apart from the fact that their argument v denotes the k parts of the virus.

Then equations 1a-1k must be adapted:

$$\exists v_1^*, \forall j, n, x, \quad v_1^*(j,n,x) = f_1^*(v_1^*, \{\mu_{i,v}\}_i, j, n, x) \tag{3a}$$

$$\cdots$$

$$\exists v_k^*, \forall j, n, x, \quad v_k^*(j,n,x) = f_k^*(v_k^*, \{\mu_{i,v}\}_i, j, n, x) \tag{3k}$$

as well as equation 2:

$$v(x) = f(v_1, \cdots, v_k, \{\mu_{i,v}\}_i, x) \tag{4}$$

Thus, we're back with a similar system as previously. Adding the equations on the $\mu_{i,v}$ – using the g_i or h_i functions –, the polyadic recursion theorem entails the existence of the v_j^* and of the $\mu_{i,v}$. Finally, using equations 3a-3k, one can ensure that equation 4 is still verified.

3.6.2 Class A viruses – dependent parts

The case of dependent parts is very similar, in its formalization, to the independent one. This merely amounts to adding a dependency of the f_i (resp. f_i^*) on all v_i (resp. v_i^*). Then, together with the equations on the mutation functions $\mu_{i,v}$, we can apply the polyadic recursion theorem, which entails existence of these functions.

The final equation must take into account these new dependencies, in the f_i^* expressions, but, as one can check, it remains verified.

Also, Filiol defined another class of combined viruses, namely the class C, which corresponds to weakly dependent codes, where the dependency only exists in one direction – v_1 is aware of v_2 but this is not true conversely. This class is a specific case of dependent parts where the function f_i (resp. f_i^*) does not depend on the parts $v_{j<i}$ (resp. $v_{j<i}^*$). In that particular case and when not considering the mutation functions, Kleene's recursion theorem can be repeatedly applied

k times – starting from the last part – in order to yield the existence of parts v_i, thanks to the special form of these equations:

$$\begin{array}{lll} \exists v_1^*, \forall j, n, x, \quad v_1^*(j,n,x) & = & f_1^*(v_1^*, \cdots, v_k^*, \{\mu_{i,v}\}_i, j, n, x) \\ & \cdots & \\ \exists v_k^*, \forall j, n, x, \quad v_k^*(j,n,x) & = & f_k^*(v_k^*, \{\mu_{i,v}\}_i, j, n, x) \end{array}$$

Theoretically speaking, class C viruses are thus, despite what we could have thought, closer to class B viruses (independent parts) than to class A viruses. This similarity actually motivated the choice of distinguishing into separate classes weakly dependent codes from strongly dependent codes.

However this property is no longer verified when considering mutation functions – as one would expect since these mutation functions strictly depend on all parts.

Finally, a particular case of such dependent viruses consists of executing only the first virus part v_1, which will in turn execute the other parts when needed. This is the behaviour of sequential class A combined viruses, which are, along with class C viruses, the most common combined viruses. This case corresponds to the following equation:

$$v(x) = f_1(v_1, \cdots, v_k, \{\mu_{i,v}\}_i, x)$$

which corresponds to a particular case of the execution sequence of v (and hence of its execution function f).

Thus, this difference between class A (dependent parts) and class B viruses (independent parts) results – when not considering the mutation functions – in a unique application of the k-ary recursion theorem, for the first case, wrt to k independent applications of the basic recursion theorem, for the second case. In a sense, "viral dimensions" are preserved in the recursion theory.

4 Formal Grammars and Recursion

Viral mutation can be modelled by formal grammars, as detailed in [Fil07]. Syntactic polymorphism can consist in transforming groups of instructions in other groups of instructions: detection of a mutated form of a virus then relies on the complexity of the associated formal grammar. Functional polymorphism can also be modelled by formal grammars, where the terminal symbols are behaviours instead of instructions. More generally, metamorphic viruses transform their code entirely. Thus, a metamorphic virus can be represented by a grammar which operates on other grammars. Filiol proposes the following definition [Fil07]:

Definition 8 (Metamorphic virus)**.** *A metamorphic virus is represented by a grammar $G = (N, T, S, R)$ where T is a set of grammars (over programs) and S is the initial grammar (first generation of the virus). Each generation of the virus corresponds to a word of a grammar G' such that $G' \in L(G)$.*

Thus, when the form v_i of a metamorphic virus represented by a grammar G replicates into a form v_{i+1}, we have:

$$v_i \Rightarrow_G v_{i+1} \text{ and } v_i \Rightarrow_{v_i} v_{i+1}$$

This definition involves that a grammar G_i associated to generation i must behave locally (on G_i) as the grammar G, since G represents the global behaviour of the virus v for any generation. Thus we perceive a first notion of recursion. Also, grammars that operate on grammars are a second, more straightforward, notion of recursion: in the recursion theory, recursive functions can indeed be seen as integers operating on integers.

Also, the equivalence between formal grammars and Turing machines gives sense to the study of recursion inside the theory of formal grammars. We first consider the example of Quine grammars which illustrates even more the interest of considering formal grammars from a recursive point of view.

4.1 Quine grammars

Quine programs are programs that exactly output their own source code. For instance, a basic trick is to define a function that outputs a string which contains a recursive reference to itself: the program calls this function with its code, replacing this very call with a recursive reference.

```
void print (char *s) {
  ...  /* this outputs s and replaces any occurence of %% by s.
}
void main () {
  print ("void print (char *s) {"
         ...
         "}"
         "void main () {"
         "  print (\"%%\");"
         "}");
}
```

Thus we can also imagine formal grammars that output – in an encoded way – their own code – meaning an unambiguous encoding of their set of rewriting rules. Such Quine grammars can follow the same algorithmic principles as for Quine programs. We give a constructive example of such a grammar in appendix A.

4.2 Recursion theorems

Existence of Quine programs comes from Kleene's recursion theorem (theorem 2), applied to function $f : x, y \mapsto x$, which entails the existence of a program p such that:

$$\forall x, \varphi_p(x) = p$$

Thus it seems legitimate to define a recursion theorem for formal grammars, given the equivalence between type 0 grammars (unrestricted grammars) and recursively enumerable languages (recognizable by Turing machines).

Theorem 4 (First Recursion Theorem)**.** *Given a formal grammar $G = (\Delta, N, T)$, there exists a grammar $G' = (\Delta', N, T)$ such that:*

$$\forall X \in (N \cup T)^*,\ \exists \alpha \in T^* \cup \infty, \quad X \xrightarrow{G'}{}^* \ \alpha \ {}^*\xleftarrow{G} \langle G', X \rangle$$

$X \xrightarrow{G}{}^* \infty$ means that X cannot rewrite into any terminal sequence (either because of an infinite sequence of rewritings, denoting a loop in a program, or because no rewriting rule can be applied). $\langle G', x \rangle$ denotes the encoded pair of a representation $[G']$ of G' and x (using some appropriate encoding).

A second recursion theorem can also be inferred:

Theorem 5 (Second Recursion Theorem)**.** *Given a formal grammar G, there exists a grammar G' such that:*

$$\forall x, \quad x \in \mathcal{L}(G') \iff \langle G', x \rangle \in \mathcal{L}(G)$$

Both theorems are direct formulations of Kleene's recursion theorem. The first theorem transforms a semi-recursive function into a grammar which rewrites an input into an output and conversely. The second theorem transforms a semi-recursive function into a grammar recognizing the words on which this function is defined and conversely (since any recursively enumerable language can be recognized by a semi-recursive function).

Then, the existence of Quine grammars comes from theorem 4 applied to the grammar G with the following rule:

$$\langle X, Y \rangle \Rightarrow X$$

We get:

$$\exists G', \forall X, \quad X \xrightarrow{G'}{}^* [G']$$

Theorem 5 could also have been used with the grammar G recognizing all couples $\langle w, w \rangle$. Thus G' recognizes only one word, which is the representation of itself $[G']$.

4.3 Iteration function

The iteration function, also called S-m-n function and denoted by S, is easily transposable to formal grammars. Consider a grammar G, that takes an input $\langle x, y \rangle$. Specialization of G for input x can be simply defined by the grammar G' that first transforms y in $\langle x, y \rangle$ and then uses the rules of G. Note that this is similar to the common programming way which would specialize $f(x, y)$ for its input x by defining the function: $g(y) = f(x, y)$. Thus this formal grammars perspective allows us to match the theory with its algorithmic counterpart.

Theorem 6 (Iteration theorem)**.** *There exists a formal grammar $\mathcal{S}$ which verifies:*

For any grammar G, for any word $X \in (N \cup T)^*$, $\mathcal{S}$ transforms $\langle G, X \rangle$ into the representation $[G']$ of a grammar G' such that:

$$\forall Y \in (N \cup T)^*,\ \exists \alpha \in T^* \cup \infty, \quad \langle X, Y \rangle \xrightarrow{G}{}^* \ \alpha \ {}^*\xleftarrow{G'} Y$$

This section has highlighted the analogy between recursive functions and formal grammars and built a bridge between abstract virology studied from the somehow semantic point of view based on the formal grammars theory and abstract virology studied from the functional point of view based on the recursion theory.

5 Discussion

Studying viruses in the frameworks of recursion theory and of formal grammars allows to identify more precisely mechanisms on which virus reproduction relies or mechanisms that it involves. While Bonfante & al. were more interested in the replication itself, we were concerned with mutation aspects that occur during this replication. Knowing these mechanisms is then helpful in the following scopes:

- Understanding the underlying stakes and logic in viral detection and protection. This is a necessary preliminary step to the next application:
- Defining new detection models in which those mechanisms are controlled and/or restricted, and studying their viability, the involved limitations, etc.
- And last but not least: identifying new threats. The recursion framework is well studied and allows, as our results demonstrated it, to discover viral behaviours that are valid from a theoretical perspective but have no equivalent (yet) in the real world. Consider for instance the case of infinite vertical mutation chains. Also, applying the current results to a new compatible framework (like a network configuration, etc.) may also appear to be useful in order to assess the security of this framework and identify threats that could plague it in a near future.

Thus, though this study might seem a bit abstract with regard to the actual antiviral defense, the theories of recursion and of formal grammars are very powerful frameworks where viral techniques can be both modelled and studied. Furthermore, studying viruses in such theories plays a key role in the necessary proactive approach that aims to identify tomorrow's evolution of today's threats.

6 Conclusion

We have extended the relation between the recursion theory and the concept of viral replication and mutation to any depth of mutation, showing by the way the theoretical existence of viruses with an infinite vertical mutation. This formalism considers a behaviour-based approach, as was done in Bonfante & al seminal work, along with a syntax-based approach which allows for more practical considerations, namely accessing the mutation functions of a virus. Also we introduced some basic notions of recursion in the theory of formal grammars: the formalization of metamorphic viruses by grammars operating on other grammars makes this approach somehow promising. Future work will investigate this new approach in regards of virus behaviour and virus detection.

We did not consider interactions in our formalism, although actual viruses tend to use it more and more: the study of combined viruses also showed the practical interest of considering such interactions. Some work has already been done to address this need, like in [JFD07]. Future work will thus try to reconcile this formalism with the theory of interactions.

References

[Adl88] Leonard M. Adleman. An abstract theory of computer viruses. In *Advances in Cryptology - CRYPTO'88*, volume 403, pages 354 – 374. Springer, 1988.

[BKM05] Guillaume Bonfante, Matthieu Kaczmarek, and Jean Yves Marion. Toward an abstract computer virology. In *Lecture Notes in Computer Science*, volume 3722, pages 579 – 593. Springer, October 2005.

[BKM07] Guillaume Bonfante, Matthieu Kaczmarek, and Jean Yves Marion. A classification of viruses through recursion theorems. In *International Workshop on the Theory of Computer Viruses*, May 2007.

[Coh86] Fred Cohen. *Computer Viruses*. PhD thesis, University of Southern California, January 1986.

[Fil05] Éric Filiol. *Computer viruses: from theory to applications*. Springer Verlag, 2005.

[Fil07] Éric Filiol. *Advanced Viral Techniques*. Springer-Verlag France, 2007. An english translation is pending.

[JFD07] Grégoire Jacob, Éric Filiol, and Hervé Debar. Malwares as interactive machines: A new framework for behavior modelling. In *2nd Workshop on the Theory of Computer Viruses*, 2007.

[Kle38] Stephen Cole Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3(4):150 – 155, December 1938.

[Smu93] Raymond Smullyan. *Recursion Theory for Metamathematics*. Oxford University Press, 1993.

[ZZ04] Zhihong Zuo and Mingtian Zhou. Some further theoretical results about computer viruses. *The Computer Journal*, 2004.

A Quine grammars

Consider the following example of a Quine program:

```
void print (char *s) {
  ...  /* this outputs s and replaces any occurence of %% by s. */
}
void main () {
  print ("void print (char *s) {"
         ...
         "}"
         "void main () {"
         "  print (\"%%\");"
         "}");
}
```

A Quine grammar can now use the same principle. Let's denote the initial non terminal symbol by S. We want our grammar G to rewrite S in a representation of G. This representation is free and should allow encoding and decoding of any grammar. We will use the following convenient representation:

- a sequence of rules $\delta_1, \cdots, \delta_n$ is represented by $[\delta_1]; [\cdots]; [\delta_n]$, where $[\delta]$ is the representation of the rule δ.
- a rule $A \Rightarrow B$ is represented by $[A] : [B]$.
- a word $X.W$ is represented by $x.[W]$, where x is a terminal symbol associated to X.

This representation actually needs a slight modification to build a Quine grammar. Let's consider a rule $A.x \Rightarrow B$, where x must match any possible non terminal symbol used by the representation of this rule. Then, we will have the rule: $A.a \Rightarrow B$ but we now need to represent a, say by a'. This requires the rule $A.a' \Rightarrow B$, $A.a'' \Rightarrow B$ and so on. To overcome this, we introduce terminal symbols n, t, s for non terminal symbols, terminal symbols and special symbols (like ; and :). Thus, we will have the following rules:

- $A.a \Rightarrow B$, represented by $na.ta : [B]$.
- $A.n \Rightarrow B$, represented by $na.tn : [B]$.
- $A.t \Rightarrow B$, represented by $na.tt : [B]$.

- $A. :\Rightarrow B$, represented by $na.s :: [B]$.
- $A.s \Rightarrow B$, represented by $na.ts : [B]$.

Such a representation allows unique encoding and decoding.

Since our grammar will work as defined by our example Quine program, we define a *print* macro, represented by the non-terminal symbol P and the special symbol $\blacklozenge$ to denote the recursive reference. P must then replace this reference by the original word, so we need to duplicate this word: $P.a.b.\blacklozenge.c.\square$ is transformed in $a.b.\lozenge'.c.\bigstar.a.b.\lozenge.c.\square$ and finally in $a.b.a.b.\lozenge.c.c$, with $\lozenge$ being the terminal representation of $\blacklozenge$.

The following rules represent the *print* macro:

- Duplication rules:
  - $P \Rightarrow \bigstar.P'$
  - $P'.x \Rightarrow\prec .x.x.P'$, foreach non terminal symbol x appearing in the final representation of these rules.
  - $P'.\square \Rightarrow \square$.
  - $P'.\blacklozenge \Rightarrow\prec .\lozenge'.\lozenge.P'$.
  - $\bigstar. \prec .y \Rightarrow y.\bigstar$ and $x. \prec .y \Rightarrow\prec .y.x$, foreach non terminal symbols x and y appearing in the final representation of these rules.
- Substitution rules:
  - $x.\bigstar.y \Rightarrow\prec .y.x.\bigstar$.
  - $x.\bigstar.\square \Rightarrow\prec .\square.x$.
  - $x. \prec .y \Rightarrow\prec .y.x$.
  - $\lozenge'. \prec .y \Rightarrow y.\lozenge'$.
  - $\lozenge'. \prec .\square \Rightarrow$.

The final rule looks like: $S \Rightarrow P.\cdots .; .ns. : .np.\blacklozenge.s.\square.\square$, where $\cdots$ contains the linear representation of the *print* macro (previous rules). This grammar will first rewrite S in $P.\cdots .; .ns. : .np.\blacklozenge.s\square.\square$, then in $\cdots .; .ns. : .np.\lozenge'.s.\square.\bigstar.\cdots .; .ns. : .np.\lozenge.s.\square.\square$ and finally in $\cdots .; .ns. : .np.\cdots .; .ns. : .np.\lozenge.s.\square.s.\square$, which will be interpreted as the P rules followed by $S \Rightarrow P.\cdots .; .S. : .P.\bigstar.s.\square.\square$.

B Recursion Theorems

B.1 Polyadic (or n-ary) Recursion Theorem

We generalize Smullyan's double recursion theorem to any number of recursive functions. First it can be extended to any finite set of semi-computable functions.

Theorem 7 (Polyadic Recursion Theorem)**.** *Let $f_1, \ldots, f_n$ be n semi-recursive functions, where $n \geq 1$. Then there exist n semi-recursive functions $e_1, \ldots, e_n$ such that:*

$$\begin{array}{rrcl} \forall x, & e_1(x) & = & f_1(e_1, \cdots, e_n, x) \\ & & \cdots & \\ & e_n(x) & = & f_n(e_1, \cdots, e_n, x) \end{array}$$

Proof. Let p, q be two semi-computable functions: $\langle p, q \rangle$ denotes the function that returns $\langle p(x), q(x) \rangle$ on an input x.

We will show this result for $n = 3$. The general case follows by an easy induction. Let f_1, f_2, f_3 be three semi-computable functions, with inputs (p, q, r, x). We define the semi-computable functions g_1 and g_2 by:

$$\begin{array}{rcl} g_1(p, \langle q, r \rangle, x) & = & f_1(p, q, r, x) \\ g_2(p, \langle q, r \rangle, x) & = & \langle f_2(p, q, r, x), f_3(p, q, r, x) \rangle \end{array}$$

Then there exists e'_1, e'_2 such that: $e'_1(x) = g_1(e'_1, e'_2, x)$ and $e'_2(x) = g_2(e'_1, e'_2, x)$. Finally we define e_1, e_2, e_3 by: $e_1 = e'_1$, $\langle e_2, e_3 \rangle = e'_2$. □

Note that this proof uses Smullyan's double recursion theorem though we could have used Kleene's recursion theorem by considering functions of $\mathbb{N} \times \mathbb{N}^n$.

B.2 Countable recursion theorem

The polyadic recursion theorem is defined for finite cases but can be extended to the countable case.

Theorem 8 (Countable Recursion Theorem)**.** *Let $\{f_i\}$ be a countable (recursive) set of semi-recursive functions. Then there exists a countable set of semi-recursive functions $\{e_i\}$, accessible through a semi-recursive function E, such that:*

$$\begin{array}{rrcl} \forall x, & e_1(x) & = & f_1(E, x) \\ & e_2(x) & = & f_2(E, x) \\ & & \cdots & \end{array}$$

Proof. Let F be the semi-recursive function such that: $\forall i, F(i) = f_i$. Then the existence of E, and hence of the corresponding e_i's, comes from the recursion theorem applied to the function $f = \langle i, x \rangle \mapsto F(i)(E, x)$. □